A Generic Object-Calculus Based on Addressed Term Rewriting Systems

Dan Dougherty
Wesleyan University
Middletown, CT 06459, USA

Frédéric Lang
INRIA Rhone-Alpes
Montbonnot, 38334 St Ismier, France

Pierre Lescanne
École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon 7, France

Luigi Liquori
École des Mines-INPL-LORIA
Parc de Saurupt, 54042 Nancy, France

Kristoffer Rose
IBM T. J. Watson Research Center
30 Saw Mill River Road, Hawthorne, NY 10532, USA

February 2, 2001

Abstract

We describe the foundations of $\lambda\mathcal{O}bj^{+a}$, a framework, or generic calculus, for modeling *object-calculi*. This framework is essentially a detailed formal operational semantics of object based languages, in the style of the Lambda Calculus of Objects. As a formalism for specification $\lambda\mathcal{O}bj^{+a}$ is arranged in *modules*, permitting a natural classification of many object-based calculi according to their features. In particular there are modules for calculi of non-mutable objects (*i.e.*, *functional object-calculi*) and for calculi of mutable objects (*i.e.*, *imperative object-calculi*). As a computational formalism $\lambda\mathcal{O}bj^{+a}$ is based on rewriting rules. Classical first-order term rewriting systems are not appropriate since we want to reflect aspects of implementation practice such as sharing, cycles in data structures and mutation. Therefore we define the notion of *addressed terms*, and develop the corresponding notion of *addressed term rewriting*.

Contents

List of Figures

1 Introduction

Recent years have seen a great deal of research aimed at providing a rigorous foundation for object-oriented programming languages. In many cases this work has taken the form of "object-calculi" [FHM94, AC96]. Here, "object-based" is to be understood as in contrast to "class-based".

Such calculi can be understood in two ways. On one hand, the formal system is a description of the semantics of the language, and can be used as a framework for classifying language design choices, to provide a setting for investigating type systems, or to support a denotational semantics.

Alternatively, we may treat an object-calculus as an intermediate language into which user code (in a high-level object-oriented language) may be translated, and from which an implementation (in machine language) may be derived. Then the main problem is to describe how to correctly and efficiently execute terms in the object-calculus from which support for implementing several high-level object-calculi can be achieved.

In this paper we present the calculus $\lambda\mathcal{O}bj^{+a}$ in which one can give a formal specification of the operational semantics for a variety of object-based programming languages. In fact, $\lambda\mathcal{O}bj^{+a}$ (introduced in [LLL99]) is a *generic framework*, leading to an easy *classification* of object-based languages and their semantics, making a clear distinction between functional and imperative languages *i.e.*, languages with non-mutable (persistent) objects and languages with mutable (ephemeral) objects.

The calculus $\lambda\mathcal{O}bj^{+a}$ is based on λ-calculus. Despite this fact, we stress that we do *not* restrict our attention to so-called "functional" object-oriented calculi. A key feature of our approach is the representation of programs as *addressed terms* [LDLR99] which support reasoning about mutation. Since $\lambda\mathcal{O}bj^{+a}$ contains the λ-calculus explicitly, we therefore have a modular and uniform treatment of both functional and imperative programming.

Many treatments of functional operational semantics exist in the literature [Lan64, Aug84, Kah87, MTH90]. To go further and accommodate imperative operations one can use the traditional *stack and store* approach [Plo81, Tof90, FH92, WF94, AC96, BF98]. The greater complexity of the presentation of the latter works might lead one to conclude—wrongly—that implementing functional languages is easy in comparison with imperative languages. Such a false impression may be due in part to the fact that typical operational semantics formalisms are based on algebras: this makes them good at abstracting away the complexity of the algebraic structures used in functional languages but ill-suited to express the non-algebraic structure of imperative data structures. The novelty of $\lambda\mathcal{O}bj^{+a}$ is that it provides an *homogeneous* approach to both functional and imperative aspects of programming languages, in the sense the two semantics are treated in the same way using addressed terms, with only a minimal sacrifice in the permitted algebraic structures.

From another point of view the use of addressed terms suggests a bridge between the operational and denotational semantics for a language. This is not a direction we pursue in detail in this paper but the main idea is as follows. A traditional denotational semantics for imperative languages involves the *store*, *i.e.* a function from locations to values, as one of the key domains. The store typically has no explicit representation in the programming language and it has no structure beyond being a function space.

Now, if we identify locations with addresses in an addressed-term representation of a program, we may see the store as intimately bound up with the program expression. Indeed, a program expression embodies a function from addresses to sub-expressions whose values in turn comprise the store. Note that the semantics now need not refer to the entire store

but only that finite part concerning addresses explicit in the program. Thus the store inherits a structure, induced by the program's tree-structure and there is now a notion of one store-location occurrence being within the scope of another.

All of this leads to a new and rather subtle relationship between operational and denotational semantics.

Our main concern is thus to find the right level of abstraction, more general and robust than the machine level, yet more concrete and operational than a purely mathematical treatment *à la* λ-calculus.

Specifically, the calculus $\lambda\mathcal{O}bj^{+a}$ enjoys the following properties:

- It is *faithful to implementation* in the sense that each transition in the system corresponds to a constant-cost operation in the execution of code on a machine. This permits reasoning about resource usage and the actual cost of certain implementation choices;
- It is *a formal system* which can support a careful analysis of some fundamental properties of object-oriented languages, such as type-safety and observational equivalence.

With regard to the first point, $\lambda\mathcal{O}bj^{+a}$ gives an explicit account of substitution, sharing, and redirection. We feel that the inclusion of explicit indirection nodes is a crucial innovation here. Indirection nodes allow us to give a more realistic treatment of the so-called collapsing rules of term graph rewriting (rules that rewrite a term to one of its proper sub-terms): more detailed discussion will be found in Sections 3.5 and 4.2.

The framework $\lambda\mathcal{O}bj^{+a}$ is much more than a simple object-calculus. It is defined in terms of a set of four *modules* (L, C, F, and I), each of which captures a particular aspect of object-calculi. Indeed, the modules are sets of rules which describe, in "small steps", the transformations of the objects, whereas the strategies (such as call-by value, call-by-name, etc) describe how these rules are invoked giving the general evolution of the whole program. Usually in the description of an operational semantics, strategies and small steps are tightly coupled. In our approach they are strongly independent. As a consequence, we get the genericity of $\lambda\mathcal{O}bj^{+a}$, in the sense that many semantics can be instantiated in our framework to conform to specific wishes. A specific calculus is therefore a combination of *modules plus a suitable strategy*. Thus we choose to not code strategies into the framework itself and we postpone discussion of specific strategies for future work.

A useful way to understand the current project is by analogy with graph-reduction as an implementation-calculus for functional programming. As such we may see $\lambda\mathcal{O}bj^{+a}$ as part of a fundamental correspondence:

Funct. Programm.	Graph rewriting and explicit substitutions
O.O. Programm.	Addressed term rewriting and explicit substitutions ($\lambda\mathcal{O}bj^{+a}$)

In the remainder of this introduction we make the analogy above more precise, describing the technical and historical context of $\lambda\mathcal{O}bj^{+a}$, before giving the outline of the rest of the paper.

1.1 Addressed Calculi and Explicit Substitutions for λ-calculi

It is well-understood that the λ-calculus [Chu41, Bar84] is of fundamental importance in understanding the semantics of both imperative and functional programming languages [Lan66, Sto77]. We are mainly interested in the role of λ-calculus in implementations.

Explicit Substitution Calculi. These were invented in order to give a finer description of the meta-operation of *substitution*, a fundamental notion in any programming language (see for instance [ACCL91, Les94, BR95]). Roughly speaking, an explicit substitution calculus fully includes the substitution operation as part of the syntax, adding suitable rewriting rules to deal with it. These calculi give a good model of the concept of "closures" that represent partially computed function applications. If combined with updating, closures can represent objects by considering the state of the object as what has been computed "so far" [ASS85].

The semantics of sharing. Efficient implementations of lazy functional languages (or of computer algebras) require some sharing mechanism to avoid multiple computations of a single argument. Term graphs [Wad71, Tur79, BVEG$^+$87, Plu99] have been studied as a representation of program-expressions intermediate between abstract syntax trees and concrete representations in memory, and term-graph rewriting provides a formal operational semantics of functional programming sensitive to sharing.

However, compared with thinking with finite terms, representing and thinking with graphs can be awkward. Indeed, one is faced with non well-founded relations which prevent proofs by induction. (Observe that graphs differ from trees in that the latter naturally support definition and proof by structural induction).

In this paper we will annotate terms (trees) with *global addresses* [FF89, Ros96]. Levy [Lév80] and Maranget [Mar92] propose using *local addresses*, but from the point of view of the operational semantics, global addresses describe better what is going in a computer or an abstract machine. With explicit global addresses we can keep track of the sharing that could be used in the implementation of a particular λ-calculus of explicit substitution. Sub-terms which share a common address represent the same sub-graphs, as suggested in Figure 1 (left), where a and b denote addresses. A specific notion of rewriting is introduced in order to rewrite simultaneously all sub-terms sharing a same address, mimicking what would happen in an actual implementation. These ideas were also presented in [BRL96] in the context of a simple λ-calculus with explicit substitution. We enrich the sharing with a special *back-*

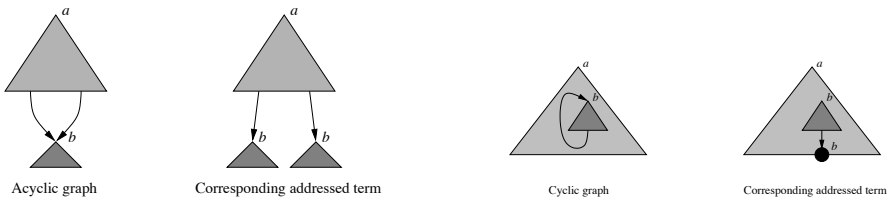

Figure 1: Representation of sharing and cycles through addresses (schematic)

pointer to handle *cyclic graphs* [Ros96]. Cycles are used in the functional language setting to represent infinite data-structures and (in some implementations) to represent recursive code; they are also interesting in the context of imperative object-oriented languages where *loops in the store* may be created by imperative updates through the use of `self` (or `this`), see Figure 6. The idea of the representation of cycles via addressed terms is rather natural: a cyclic path in a finite graph is fully determined by a prefix path ended by a "jump" to some node of the prefix path (represented with a back-pointer), as suggested in Figure 1 (right).

In [LDLR99], addressed terms are studied in the context of *addressed term rewriting*, as an extension of classical first-order term rewriting. The notion of computation on terms is expanded to encompass computations performing mutation, still through rewriting rules.

It is natural to apply these techniques in the setting of object-oriented languages to the notion of destructive updates, or update of the value of a field (as for instance the increment of a counter encapsulated in an object).

1.2 Object-based Languages

Recent years have seen the development of *object-based* languages (see [AC96] Chapter 4). Object-based languages can be either viewed as a novel object-oriented style of programming (such as in Self [US87], Obliq [Car95], Kevo [Tai92], Cecil [Cha93], and O-{1,2,3} [AC96]). In object-based languages there is no notion of class: the inheritance takes place at the object level. Objects are built "from scratch" or by inheriting the methods and fields from other objects (sometimes called *prototypes*).

Two classical problems we find in the literature regarding the implementation of imperative and flexible object-calculi are:

- The capacity to handle *loops in the store* [AC96];
- The capacity to dynamically extend objects.

The latter feature is usually forbidden when one wants to have a sound type system (the imperative object-calculus of [BF98] uses a "functional" extension).

Among the proposals firmly setting the theoretical foundation of object-based languages, the *Lambda Calculus of Objects* ($\lambda\mathcal{O}bj$) of Fisher, Honsell, and Mitchell [FHM94] has formed one of the two major schools of calculi for modeling object-oriented programming (the other is the *Object Calculus* of Abadi and Cardelli [AC96]).

$\lambda\mathcal{O}bj$ is an untyped λ-calculus enriched with object primitives. Objects are *untyped* and a new object can be created by modifying and/or extending an existing prototype object. The result is a new object which inherits all the methods and fields of the prototype. The consistency of dynamic object-extension with a sound type-system was one of the main goals of $\lambda\mathcal{O}bj$. This calculus is computationally complete, since the λ-calculus is built in the calculus itself.

The calculus $\lambda\mathcal{O}bj^{+}$ [GHL98] is an extension of $\lambda\mathcal{O}bj$ with a type system and a type soundness result ensuring that a typed program "cannot go wrong". In particular, $\lambda\mathcal{O}bj^{+}$ allows typed objects to extend themselves upon the reception of a message.

1.3 Road Map

This brings us to the $\lambda\mathcal{O}bj^{+a}$ framework, the subject of this paper. $\lambda\mathcal{O}bj^{+a}$ is based on $\lambda\mathcal{O}bj^{+}$, and uses addressed terms and explicit substitution. It is faithful to mainstream object-based programming languages in the sense that an object-calculus represents a core calculus to analyze these languages, and it extends traditional graph-reduction techniques by expressing the basic computational steps of object-oriented programming, including message-passing, method update, and especially object mutation.

In Section 2, we discuss by an example the main concepts that a generic calculus of objects has to take into account. In Section 3, we say how addressed term rewriting systems give solutions to the basic questions of object oriented languages, namely sharing, cycles

and mutations. Section 4 presents the four modules of rewriting rules that form the core of $\lambda\mathcal{O}bj^{+a}$. Section 5 concludes and describes related and further works. For pedagogical reason we put in the final Appendix the formal treatment of addressed term rewriting sytems (ATRS).

2 A Simple Example Exploiting Object Inheritance

The examples in this section embody certain choices about language design and implementation (such as deep *vs.* shallow copying, management of run-time storage, and so forth). It is important to stress that these choices are not bound up with the particular formal calculus $\lambda\mathcal{O}bj^{+a}$ which is the subject of this paper. Indeed, our main point is that $\lambda\mathcal{O}bj^{+a}$ provides a foundation for a wide variety of language styles and language implementations. We hope that the examples are suggestive enough that it will be intuitively clear how to accommodate other design choices. The main body of the paper justifies that intuition.

Those schematic examples will be also useful to understand how objects are represented and how inheritance can be implemented in $\lambda\mathcal{O}bj^{+a}$. For the sake of simplicity, we will not raise issues like privacy or encapsulation (so that we consider methods and fields to belong to the same abstraction level).

Reflecting implementation practice, in $\lambda\mathcal{O}bj^{+a}$ we distinguish two distinct aspects of an object:

- **The object structure:** the actual list of methods/fields;
- **The object identity:** a pointer to the object structure.

We shall use the word "pointer" where others use "handle" or "reference". Objects can be bound to identifiers as "nicknames" (*e.g.*, `pixel`), but that the only proper name of an object is its object identity: an object may have several nicknames (aliases) but only one identity.

Consider the following (untyped) definition of a "pixel" prototype with three fields and one method. With a slight abuse of notation, we use `:=` for both assignment of an expression to a variable, or the extension of an object with a new field or method and for overriding an existing field or method inside an object with a new value or body, respectively.

```
pixel = object {
  x := 0;
  y := 0;
  onoff := true;
  set := (a,b,c){ x := a; y := b; onoff := c; };
}
```

After instantiation, the object `pixel` is located at an address, say a, and its object structure starts at address b. See Figure 2. In what follows, we will derive three other objects from `pixel` and discuss the variations of how this may be done below.

2.1 Cloning

The first two derived objects, nick-named `p` and `q`, are *clones* of `pixel`:

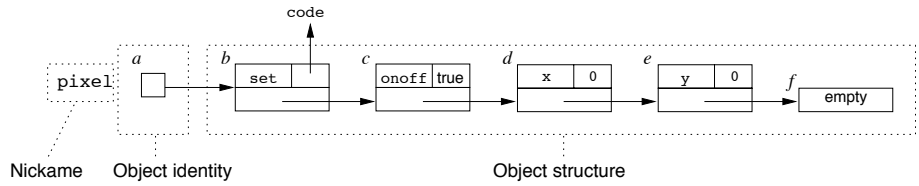

Figure 2: The Object Pixel

```
p := clone(pixel);
q := clone(p);
```

Object `p` share the same object-structure as `pixel` but of course is has its own object-identity. Object `q` share also the same object-structure of `pixel`, even if it is a clone of `p`. The effect is pictured in Figure 3. The semantics of the `clone` operator we illustrate here

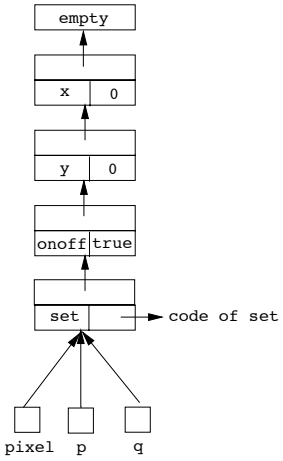

Figure 3: The Clones `p` and `q`

differs somewhat from that found in certain existing object-oriented programming languages like SmallTalk and Java. For example Java specifies that there is no sharing between the contents of the original object and its clone: the clone is a true *deep copy* of the original, but note that in Java there will be sharing between an object and its clone if an instance field of the original is itself a reference to an object. In Java a true "deep" clone of an object is in general not available via the built-in clone method, but will be provided by the programmer overriding the default method.

The `clone` operator (used by $\lambda\mathcal{O}bj^{+a}$) always produces a "shallow" copy of the prototype. As will be shown in the next subsection, however, one should not view this as simple aliasing.

In the rest of this section, we discuss the differences between functional and imperative models of object-calculi.

2.2 Illustrating an Imperative Calculus

Imperative object-calculi have been shown to be fundamental in describing implementations of class-based languages. They are also essential as foundations of object-based programming languages like Obliq and Self. The main goal when one tries to define the semantics of an imperative object-based language is to say how an object can be modified while maintaining its object-identity. Particular attention must be paid to this when dealing with object extension. The semantics of the imperative update operation is subtle because of side-effects.

Here we show what we want to model in our framework when we *override* the `set` method of the clone `q` of `pixel`, and we extend a clone `r` of (the modified) `q` with a new method `switch`.

```
q.set := (a,b,c){ x := x*a; y := y*b; onoff := c;}; (1)
r := clone(q);                                      (2)
r.switch := (){ onoff := not(onoff);};              (3)
```

Note that we have used a Java-like imperative syntax here to save parentheses.

Figure 4 shows the state of the memory after the execution of the instructions `(1,2)`. Note that after `(1)` the object `q` refers to a new object-structure, obtained by chaining the new body for `set` with the old object-structure. As such when the overridden `set` method is invoked, thanks to dynamic binding, the newer body will executed since it will hide the older one.

Observe that the override of the `set` method does not produce any side-effect on `p` and `pixel`; in fact, the code for `set` used by `pixel` and `p` will be just as before. Therefore, `(1)` only change the object-structure of `q` without changing its object-identity. This is the sense in which our `clone` operator really does implement shallow copying rather than aliasing, even though there is no duplication of object-structure at the time that `clone` is evaluated.

This implementations model, although space consuming, performs side effects in a very restricted and controlled way. Figure 5, finally, shows the final state of memory after the

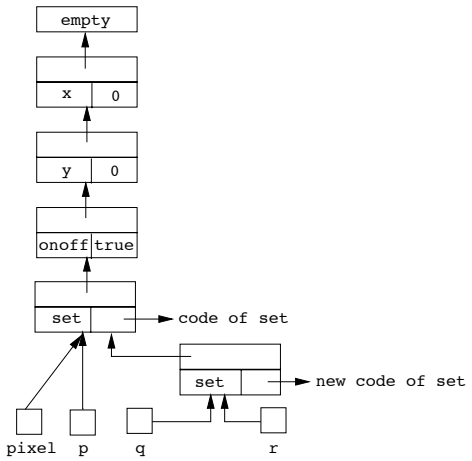

Figure 4: The Memory Structure After `(1,2)`

execution of the instruction `(3)`. Observe that in this case the update operation, denoted

by ":=", extend the object `r` with the `onoff` method. Again, the addition of the `switch`

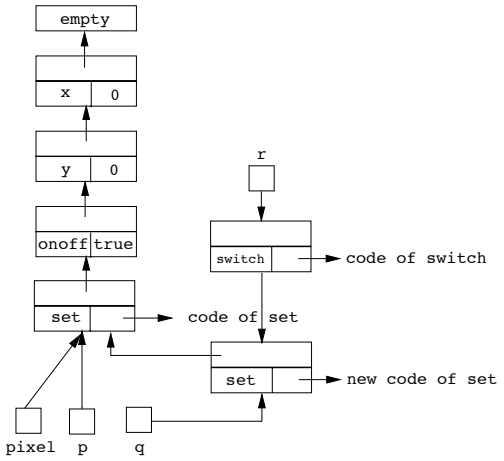

Figure 5: The Memory Structure after (3)

method change only the object-structure of `r`.

In general, changing the nature of an object dynamically by adding a method or a field can be implemented by moving the object identity toward the new method/field (represented by a piece of code or a memory location) and to *chain it* to the original structure. This mechanism is used systematically also for method/field overriding but in practice (for optimization purposes) can be relaxed for field overriding, where a more efficient *field look up and replacement* technique can be adopted. See for example the case of the Imperative Object Calculus ([AC96], Chapter 10), or observe that Java uses *static field lookup* to make the position of each field constant in the object.

This implementation model, hoverer does not avoid the unfortunate *loop in the store*: an example of loop will be given Section 3.

2.3 Illustrating a Functional Calculus

Object-calculi can play a role as well in a purely functional setting, where there is no notion of *mutable state*. As said before, an "update" operation, denoted by ":=" can either override or extend an object with some fields or methods. In a functional setting, the update *always* produces another object with its proper object-identity since this ensures that all references to an object have the same meaning whether their evaluation is delayed or not. This property is also known as *referential transparency*. Thus, the result of an update must be a fresh object in the sense that it has a proper (new) object-identity.

Probably, the purists of functional languages would not like the "imperative" notation of the *pixel,p,q,r* example; note that the definition of `pixel,p,q,r`, for example, could have been written like the following in a more traditional functional object-calculus where `let x = A in B` is syntactic sugar for the functional application `(`λ`x.B) A` and the `clone(_)` function is essentially the identity (since we have no aliasing and applications always produces new objects):

```
let p = clone(pixel) in p in
```

```
let q = clone(p).set := (a,b,c)
            {((self.x := self.x*a).y := self.y*b).onoff := c } in
  let r = (q.switch := (){ self.onoff := self.not(onoff);}) in r
```

which obviously reduces to:

```
(clone(pixel).set := (a,b,c) {...}).switch := (){...}
```

Worthy noticing is that the above code would be implemented, in a purely functional calculus, in the same way as Figure 5. This is sound since the way we treat imperative features of object-calculi is quite restricted and controlled, hence similar to the functional treatment of objects.

3 Addressed Term Rewriting Systems

The paradigm of *term rewriting* [DJ90, Klo90, BN98] is a very convenient and powerful tool to describe the operational semantics of simple calculi. In particular, term rewriting provides a computational interpretation of first-order equational reasoning.

In addition, term rewriting systems are sufficiently flexible to model the operational semantics of functional programs, although at a high level, ignoring certain aspects of memory management, reduction strategy, and parameter-passing. They are widely used to formalize, prototype, and verify software.

However, as suggested in the introduction, classical term rewriting cannot easily express issues of sharing (including cyclic data), mutation, and reduction strategies. Calculi which give an account of memory management usually introduce some *ad-hoc* data-structure to model the memory, called *heap*, or *store*, together with access and update operations. However, the use of these structures necessitates restricting the calculus to a particular strategy. The aim of addressed term rewriting systems, as the computational foundation of $\lambda\mathcal{O}bj^{+a}$, is to abstract out the notion of store so that we recover the flexibility of term rewriting, and permit description of computations in a store, by the way of *addressed rewriting rules* and the subsequent notion of rewriting.

In this section we introduce *addressed term rewriting systems* (ATRS's) informally—in the context of object-oriented programming—by examining these issues in turn and the ways in which they are reflected in features of ATRS's. To ease the reading of the paper, and for pedagogical reasons, a formal definition of addressed term and addressed term rewriting systems can be found in Appendix A.

3.1 Sharing

Sharing has been extensively studied in the context of obtaining implementations of lazy functional programming languages [PJ87, PvE93], and the initial studies of sharing in the notations of *term graph rewriting systems* [BVEG+87, Plu99], were indeed motivated by this application.

Sharing of computation. Consider the function square defined by

$$\mathsf{square}(x) = \mathsf{times}(x, x)$$

It is clear that an implementation of this function should not duplicate its input x in the expression $\mathsf{times}(x, x)$, but optimize this by only copying a *pointer* to the input. This not

only saves memory but also makes it possible to *share future computations* on x, in particular when x is not already required to be a value, as in *e.g.*, lazy programming languages. Classical term representations do not permit us to express this sharing of the actual structure of x. However, the memory structures used for the computation of a program can be represented using addressed terms. For instance, the "program" $\mathsf{square}(\mathsf{square}(2))$ can be first instantiated in memory, provided locations a, b, c to each of its constructors, as the addressed term (or memory structure) $\mathsf{square}^a(\mathsf{square}^b(2^c))$. It can then be reduced as follows:

$$\begin{aligned}
\mathsf{square}^a(\mathsf{square}^b(2^c)) &\to \mathsf{times}^a(\mathsf{square}^b(2^c), \mathsf{square}^b(2^c)) \\
&\to \mathsf{times}^a(\mathsf{times}^b(2^c, 2^c), \mathsf{times}^b(2^c, 2^c)) \\
&\to \mathsf{times}^a(4^b, 4^b) \\
&\to 16^a,
\end{aligned}$$

where "$\to$" designates one step of shared computation (we are assuming that definitions to compute the function $\mathsf{times}(x, y)$ to the value $x \times y$ exist for each x and y). The key point of a shared computation is that *all* terms which share a common address are reduced *simultaneously*. This corresponds to a *single computation step* on a small component of the memory.

Sharing of memory structures. In object-oriented programming, the aim of sharing is not only to share *computations* as in the former example, but also to share *structures*. Indeed, objects are typically structures which receive multiple pointers. Moreover, the delegation-based model of inheritance insists that object structures are shared between objects with different identities. Representing object structures with the constructors $\langle\,\rangle$ (the empty object), and $\langle \_ \leftarrow \_ \rangle$ (the functional *cons* of an object with a method/field), and object identities by the bracketing symbol $[\![\_]\!]$, the object `pixel` (of Figure 2) and the object `q` (of Figure 3) will be represented by the following addressed terms:

$$\begin{aligned}
\texttt{pixel} &\equiv [\![\, \langle\langle\langle\langle\,\rangle^f \leftarrow \texttt{y} = 0\rangle^e \leftarrow \texttt{x} = 0\rangle^d \leftarrow \texttt{onoff} = \texttt{true}\rangle^c \leftarrow \texttt{set} = \cdots\rangle^b]\!]^a \\
\texttt{q} &\equiv [\![\langle\langle\langle\langle\langle\,\rangle^f \leftarrow \texttt{y} = 0\rangle^e \leftarrow \texttt{x} = 0\rangle^d \leftarrow \texttt{onoff} = \texttt{true}\rangle^c \leftarrow \texttt{set} = \cdots\rangle^b \leftarrow \texttt{set} = \cdots\rangle^g]\!]^h
\end{aligned}$$

The use of the same addresses b, c, d, e, f in `q` as in `pixel` denotes the sharing between both object structures while g, h, are unshared locations.

3.2 Cycles

Cycles are essential in functional programming when one wants to deal with infinite data-structures in an efficient way, as is the case in lazy functional programming languages. Cycles are also used, in some implementations, to save space in the code of recursive functions.

In the context of object programming languages, cycles can be also used to express *loops* introduced in the memory (the *store*) by the imperative operators. Let us look on an example how ATRS's deal with cycles.

Example 1 (Loop in the store, [AC96]). Consider an object `o` which contains one single method, namely `m`. The method `m` overrides itself. In $\lambda\mathcal{O}bj^{+a}$, we represent methods with λ-abstractions (λ`s`.*body*) whose first parameter `s` denotes the self variable. The object `o` is

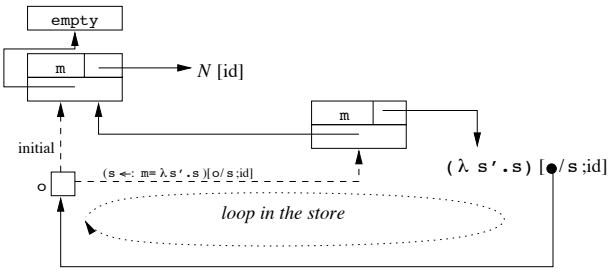

Figure 6: A Loop in the Store

then:

$$[\![\langle\langle\rangle^b \leftarrow \mathtt{m} = \underbrace{(\lambda \mathtt{s}.\langle \mathtt{s} \leftarrow: \mathtt{m} = \lambda \mathtt{s}'.\mathtt{s}\rangle)}_{N}[\mathrm{id}]^c\rangle^d]\!]^a$$

where $\langle \_ \leftarrow: \_ \rangle$ denote the imperative *cons* of an object with a method/field. For pedagogical reasons, the precise sense of [id] (the local environment of the method m) may be ignored for the moment, it will be explained later. When m is invoked on the object o, it "self-inflict" an overrides of m with a new body in which s is now bound to o itself. The result of this operation could be expressed as the *infinite term* defined by the fixed point equation:

$$\mathtt{o} = [\![\langle\langle\rangle^b \leftarrow \mathtt{m} = N[\mathrm{id}]^c\rangle^d \leftarrow \mathtt{m} = (\lambda \mathtt{s}'.\mathtt{s})[\mathtt{o}/\mathtt{s};\mathrm{id}]^e\rangle^f]\!]^a$$

Here, [o/s;id] says that in the λ-abstraction $(\lambda \mathtt{s}'.\mathtt{s})$, the free variable s is bound to o.

This is not the ATRS approach: rather than having to deal with an infinite term (or a fixed point equation), ATRS's use a *back-pointer* (noted by $\bullet$) labeled with the same address as o. The following term is the addressed term representing o, in which $\bullet^a$ denotes a pointer to the addressed term at location a, namely o itself:

$$[\![\langle\langle\rangle^b \leftarrow \mathtt{m} = N[\mathrm{id}]^c\rangle^d \leftarrow \mathtt{m} = (\lambda \mathtt{s}'.\mathtt{s})[\bullet^a/\mathtt{s};\mathrm{id}]^e\rangle^f]\!]^a$$

Figure 6 gives a graphical illustration of this example.

3.3 Mutation

Almost all object-oriented programming languages are not purely functional, but rather have some operations that may alter the state of objects without changing object's identity. An example of such mutation is given in Section 2.2 (Figures 4 and 5), where we see that the object denoted by p has had its structure altered by the addition of some methods, without changing its object identity. Note that the object p may be shared, and that all the expressions containing pointers to p undergo the mutation that happened at address a.

The key point of mutation in the setting of ATRS is the possibility to modify *in-place* the contents of any node at a given location, with respect to some precise rules. More details on computations performing mutation will be given in Section 4.

3.4 Syntax of Addressed Terms

An *addressed preterm* is a tree where each node receives a label (the operator symbol *e.g.*, times) and an address (*e.g.*, a). Of course we will not want to treat an arbitrary preterm as an addressed term, because an unconstrained preterm may denote a non-coherent memory structure. Roughly speaking, since each node in a memory has a *unique symbol* and an *unique list* of successor locations, it must be the case that all "sub-terms" at a given address in a term denote a *unique memory sub-structure*.

For instance, the instantiated "program" $\mathsf{times}^a(\mathsf{square}^b(2^c), \mathsf{square}^b(2^d))$ (quite similar to the one presented in Subsection 3.1) is not admissible, because it designates that the sons of the node at address b are both at addresses c and d. In contrast, $\mathsf{times}^a(\mathsf{square}^b(2^c), \mathsf{square}^b(2^c))$ is admissible. The Appendix provides a formal definition of addressed (pre)terms.

Above we used the word *sub-term*, but in the presence of back-pointers, this notion does not really make sense. Indeed let t be an addressed term at address a containing a back-pointer $\bullet^a$; if one takes a sub-term of t in the usual sense, say at address b, one obtains a term with a *dangling* $\bullet^a$. Therefore when defining the term at address b, which we write $t@b$, one has to expand $\bullet^a$ to avoid dangling pointers. Then, we do not call $t@b$ a "sub-term", but because of this surgery, we call it an *in-term*. Note that the relation "to be an in-term of" is not well-founded.

3.5 Rewriting

Addressed terms themselves formalize the data structures representing code and data in a machine; we now formalize the operational semantics of the machine as rewriting of addressed terms.

A rewriting rule, denoted by $l \to r$, is a pair of addressed terms with variables. As for ordinary terms, such a rule induces a reduction relation on the set of addressed terms. This relation is defined with the help of a notion of a term *matching* another term. Roughly speaking, a term t matches l when the variables of l can be substituted by addressed terms, and its addresses by other addresses, resulting in t. A precise notion of substitution is also given in Appendix: this definition is different from the usual substitution of terms, in particular in the presence of cycles. However, the intuition that substitution on addressed terms is almost the same as classical term substitution is sufficient to understand the following idea.

In an addressed rewriting rule $l \to r$, l and r must have a common address, say a, at their respective roots. The idea is that the rule describes how the node at this address has to be modified for computation. Other addresses reachable from a may be modified as well, and new nodes introduced by r.

Intuitively, given an addressed term t, the rewriting takes the following four steps:

1. *Find a redex in t i.e.*, an in-term *matching* the left-hand side of a rule.
2. *Create fresh addresses*, *i.e.*, addresses not used in the current addressed term t, which will correspond to the locations occurring in the right-hand side, but not in the left-hand side (*i.e.*, the new locations);
3. *Substitute the variables and addresses* of the right-hand side of the rule by their new values, as assigned by the matching of the left-hand side or created as fresh addresses. Let us call this new addressed term u;

4. For all a that occur both in t and u, replace in t the in-terms at address a by the in-term at address a in u.

An important constraint of ATRS's is that both members of a rule must have the same address. Hence, one rejects rules like $F^a(X) \to X$, called *collapsing* or *projection* rules. We can recover the effect of such rule by adding to the signature a unary function symbol, intuitively seen as an *indirection node* and written $\lceil\ \rceil$. This constraint is realistic as, in fact, it turns to be the technique used by all actual implementations to avoid searching the memory for pointers to redirect. With that, the above rule can be expressed as $F^a(X) \to \lceil X \rceil^a$. The use of explicit indirection nodes is motivated by our wish to explicit the constraints that every rewriting step must be as close as possible to what happens in a real implementation, so that the complexity of the rewriting and the complexity of the execution on a real machine are closely correlated. It is a simple and efficient way to avoid an unbounded, global, redirection.

The last operation in the above definition of rewriting corresponds to the simulation of updates *in-place* in a memory: all over the rewritten term, address contents are modified to give an account of the sharing and mutation. This operation is the key point of the following property: any reduction starting from an addressed term results in an addressed term *i.e.*, the coherence of the underlying memory structures is preserved by the application of any rule. Formally we have the following theorem.

Theorem 1. *Let R be an addressed term rewriting system and t be an addressed term. If $t \to u$ in R then u is also an addressed term.*

A formal definition of ATRS's, and the proof of the theorem, are given in Appendix. Section 4 gives an intuition of the way rules are defined and used, in the particular setting of $\lambda\mathcal{O}bj^{+a}$. See especially Section 4.3 in which some typical computations in $\lambda\mathcal{O}bj^{+a}$ are modeled by (addressed) rewriting.

4 Modules and Rules of $\lambda\mathcal{O}bj^{+a}$

The purpose of this section is to describe the rules of the framework $\lambda\mathcal{O}bj^{+a}$. As advertised in the title of this paper, $\lambda\mathcal{O}bj^{+a}$ is a framework described by a set of rules arranged in *modules*. The four modules are called respectively L, C, F, and I.

L is the *functional* module, and is essentially the calculus $\lambda\sigma^a_w$ of [BRL96]. This module alone defines the core of a purely functional programming language based on λ-calculus and weak reduction.

C is the *common object* module, and contains all the rules common to all instances of object calculi defined from $\lambda\mathcal{O}bj^{+a}$. It contains rules for instantiation of objects and invocation of methods.

F is the module of *functional update*, containing the rules needed to implement object update that also changes object identities.

I is the module of *imperative update*, containing the rules needed to implement object update that does not change object identities. It also provides a dynamic semantics of the `clone` operator.

The set of rules $\mathsf{L}+\mathsf{C}+\mathsf{F}$ is the instance of $\lambda\mathcal{O}bj^{+a}$ for non-mutable object calculi while $\mathsf{L}+\mathsf{C}+\mathsf{I}$ is for mutable object calculi[1]. We summarize this correspondence and the relationship between $\lambda\mathcal{O}bj^{+a}$ and classical operational semantics in the following table:

	lambda	lambda + object
functional	graph-rewriting	$\lambda\mathcal{O}bj^{+a}$ $(\mathsf{L}+\mathsf{C}+\mathsf{F})$
imperative	stack & store	$\lambda\mathcal{O}bj^{+a}$ $(\mathsf{L}+\mathsf{C}+\mathsf{I})$

We first introduce the syntax of $\lambda\mathcal{O}bj^{+a}$, then explain the rules.

4.1 Syntax of $\lambda\mathcal{O}bj^{+a}$

The syntax of $\lambda\mathcal{O}bj^{+a}$ is summarized in Figure 7. The first category of expressions is the *code* of programs. Code contains all the constructs of the calculus $\lambda\mathcal{O}bj^{+}$ [GHL98], plus an imperative update and a clone operator. Terms that define the code have no addresses, because code contains no environment and is not subject to any change during the computation (remember that addresses are meant to tell the computing engine which parts of the computation structure can change simultaneously). The second and third categories define dynamic entities, or inner structures: the *evaluation contexts*, and the *internal structure of objects* (or simply *object structures*). Terms in these two categories have explicit addresses. The last category defines *substitutions* also called *environments* *i.e.*, lists of terms bound to variables, which are to be distributed and augmented over the code.

$$
\begin{array}{rll}
M,N ::= & \lambda x.M \mid MN \mid x \mid c \mid \langle\,\rangle \mid \langle M \leftarrow m = N\rangle \mid & \text{(Code)}\\
& M \Leftarrow m \mid \langle M \leftarrow\!: \; m = N\rangle \mid \mathsf{clone}(x) & \\
U,V ::= & M[s]^a \mid (UV)^a \mid & \text{(Evaluation Contexts)}\\
& (U \Leftarrow m)^a \mid \langle U \leftarrow m = V\rangle^a \mid & \\
& \langle U \leftarrow\!: \; m = V\rangle^a \mid [\![O]\!]^a \mid \mathit{Sel}^a(O,m,U) \mid & \\
& \lceil U \rceil^a \mid \bullet^a & \\
O ::= & \langle\,\rangle^a \mid \langle O \leftarrow m = V\rangle^a \mid \bullet^a & \text{(Object structures)}\\
s ::= & U/x; s \mid \mathrm{id} & \text{(Substitutions)}
\end{array}
$$

Figure 7: The Syntax of $\lambda\mathcal{O}bj^{+a}$

The code category. Code terms, written M and N, provide the following constructs:

- Pure λ-terms, constructed from abstractions, applications, variables, and constants. This allows the definition of higher-order functions.

[1] Okasaki [Oka98] calls mutable objects *persistent* and non-mutable objects *ephemeral*.

- Objects, constructed from the empty object $\langle\rangle$ and update operators: the functional $\langle \_ \leftarrow \_ \rangle$ and the imperative $\langle \_ \leftarrow\!: \_ \rangle$. An informal semantics of the update operators has been given in Section 2. As in [GHL98], these operators can be understood as extension as well as override operators, since an override is handled as a particular case of extension.
- Method invocation $(\_ \Leftarrow \_)$.
- Cloning. $\mathsf{clone}(x)$ is an operator which creates a new object identity for the object pointed by x but still shares the same object structure as the object x itself (it is "shallow" as discussed in Section 2).

Evaluation contexts. These terms, written U and V, model *states of abstract machines*. Evaluation contexts contain an abstraction of the temporary structure needed to compute the result of an operation. They are given addresses as they denote dynamically instantiated data structures; they always denote a term closed under the distribution of an environment. There are the following evaluation contexts:

- *Closures*, of the form $M[s]^a$, are pairs of a code and an environment. Roughly speaking, s is a list of evaluation contexts that must replace the free variables in the code M.
- *Objects*, of the form $[\![O]\!]^a$, represent evaluated objects whose internal object structure is O and whose object identity is a. In other words, the address a plays the role of an *entry point* or *handle* to the object structure O, as illustrated by Figure 2.
- The terms $(UV)^a$, $(U \Leftarrow m)^a$, $\langle U \leftarrow m = V\rangle^a$, and $\langle U \leftarrow\!: m = V\rangle^a$, are the evaluation contexts associated with the corresponding code constructors. Direct subterms of these evaluation contexts are themselves evaluation contexts instead of code.
- The term $Sel^a(O, m, U)$ is the evaluation context associated to a method-lookup *i.e.*, the scanning of the object structure O to find the method m, and apply it to the object U. It is an auxiliary operator invoked when one sends a message to an object.
- The term $\lceil U \rceil^a$ denotes an indirection from the address a to the root of the addressed term U.
- The term $\bullet^a$ is a *back-pointer* intended to denote cycles as explained in Sections 1.1 and 3.

Internal Objects. The crucial choice of $\lambda\mathcal{O}bj^{+a}$ is the use of *internal objects*, written O, to model object structures in memory. They are persistent structures which may only be accessed through the address of an object, denoted by a in $[\![O]\!]^a$, and are never destroyed nor modified (but eventually removed by a garbage collector in implementations, of course). Since our calculus is inherently delegation-based, objects are implemented as linked lists (of fields/methods), but a more efficient array structure can be envisaged. Again, the potential presence of cycles means that object structures can contain occurrences of back-pointers $\bullet^a$.

The evaluation of a program, *i.e.* a code term M, always starts in an empty environment, *i.e.* as a closure $M[\mathrm{id}]^a$.

4.2 Architecture of $\lambda\mathcal{O}bj^{+a}$

The rules of $\lambda\mathcal{O}bj^{+a}$ as a computational-engine are defined in Figure 8. We will explain the rules, module by module.

The Module L. The module L is very similar to the calculus $\lambda\sigma_w^a$ of [BRL96], a calculus of explicit substitution enriched with addresses, to which we have added explicit indirections. Module L hence defines the core of a very simple functional programming language.

Rule (App) tells how environments have to be distributed over applications: it creates two new evaluation contexts (closures) located at new fresh addresses b and c; each of these closures is reachable from address a, updated so as to contain an evaluation context of application. Note that the two occurrences of s in the right-hand side of the rule contain the same addressed sub-terms. This means that these sub-terms are shared.

Once a substitution reaches an abstraction, a redex can be contracted by applying rule (Bw)[2]. This extends the substitution by adding a pair, binding the parameter of the abstraction to the argument of the application.

Once a variable is reached by a substitution, a lookup has to be performed in the substitution to find the evaluation context to be substituted *i.e.*, the one bound to the variable. This is described by rules (FVar), and (RVar). Note that, since modifications *must* be performed in place, and since U has its own address, the only simple way to get access to U from a is to set an indirection (denoted by a pair of $\lceil\ \rceil$-brackets) from a to the root of U.

The last two rules (AppRed), and (LCop) say how to get rid of indirections that could "block" the identification of redexes. Intuitively, we are here treating the situation in which address a "really" has a redex, but one of its components is available only by following a redirection. In this module, an indirection blocks a reduction if the indirected node is an abstraction, and the indirection node is the left argument of an application. We have two alternative ways to get rid of such indirections, modeling choices that may be made in an implementation; a similar discussion can be found in [BRL96]:

1. Redirect from the address a to the root of U as in rule (AppRed);
2. Copy the indirected abstraction node lying at address b, at the address of the indirection node a, as in rule (LCop). Note that the copy is only a copy of the node at b, not of a whole graph, since addresses in s and (implicit addresses) in $\lambda x.M$ do not change. Note as well that this copy may not cause a loss in the sharing of computation since an abstraction is already a value and can not be reduced further.

Finally observe that in contrast to [BRL96], no rule is given which allows us to copy shared structures for applications and other closures. There are two reasons to do this: the first is that it could induce a *loss in the sharing* of computations since applications and closures are not values; the second (stronger) is that such closures can reduce to objects, and, as we will see later, a copy would have the same effect as a *clone of object*. We certainly do not want to have uncontrolled cloning of objects, particularly in the presence of imperative update. In fact, the way an implementer is going to handle redirections is an essential component of the design of an object oriented language. One main purpose of our approach is to make this pointer manipulation explicit in a rewriting framework.

[2]The name (Bw) comes from "Beta weak", the name this rule is assigned in functional languages. A more appropriate pronunciation would be "bind weakly".

The Module L

$$(MN)[s]^a \to (M[s]^b\, N[s]^c)^a \quad \text{(App)}$$
$$\big((\lambda x.M)[s]^b\, U\big)^a \to M[U/x;s]^a \quad \text{(Bw)}$$
$$x[U/y;s]^a \to x[s]^a \quad x \not\equiv y \quad \text{(RVar)}$$
$$x[U/x;s]^a \to \lceil U \rceil^a \quad \text{(FVar)}$$
$$(\lceil U \rceil^b\, V)^a \to (U\, V)^a \quad \text{(AppRed)}$$
$$\lceil (\lambda x.M)[s]^b \rceil^a \to (\lambda x.M)[s]^a \quad \text{(LCop)}$$

The Module C

$$\langle\rangle[s]^a \to \llbracket \langle\rangle^b \rrbracket^a \quad \text{(NO)}$$
$$(M \Leftarrow m)[s]^a \to (M[s]^b \Leftarrow m)^a \quad \text{(SP)}$$
$$(\llbracket O \rrbracket^b \Leftarrow m)^a \to Sel^a(O, m, \llbracket O \rrbracket^b) \quad \text{(SA)}$$
$$(\lceil U \rceil^b \Leftarrow m)^a \to (U \Leftarrow m)^a \quad \text{(SRed)}$$
$$Sel^a(\langle O \leftarrow m = U \rangle^b, m, V) \to (U\, V)^a \quad \text{(SU)}$$
$$Sel^a(\langle O \leftarrow n = U \rangle^b, m, V) \to Sel^a(O, m, V) \quad m \not\equiv n \quad \text{(NE)}$$

The Module F

$$\langle M \leftarrow m = N \rangle[s]^a \to \langle M[s]^b \leftarrow m = N[s]^c \rangle^a \quad \text{(FP)}$$
$$\langle \llbracket O \rrbracket^b \leftarrow m = V \rangle^a \to \llbracket \langle O \leftarrow m = V \rangle^c \rrbracket^a \quad \text{(FC)}$$
$$\langle \lceil U \rceil^b \leftarrow m = V \rangle^a \to \langle U \leftarrow m = V \rangle^a \quad \text{(FRed)}$$

The Module I

$$\langle M \leftarrow\!: m = N \rangle[s]^a \to \langle M[s]^b \leftarrow\!: m = N[s]^c \rangle^a \quad \text{(IP)}$$
$$\langle \llbracket O \rrbracket^b \leftarrow\!: m = V \rangle^a \to \lceil \llbracket \langle O \leftarrow m = V \rangle^c \rrbracket^b \rceil^a \quad \text{(IC)}$$
$$\langle \lceil U \rceil^b \leftarrow\!: m = V \rangle^a \to \langle U \leftarrow\!: m = V \rangle^a \quad \text{(IRed)}$$
$$\mathsf{clone}(x)[U/y;s]^a \to \mathsf{clone}(x)[s]^a \quad x \not\equiv y \quad \text{(SRVar)}$$
$$\mathsf{clone}(x)[\llbracket O \rrbracket^b / x; s]^a \to \llbracket O \rrbracket^a \quad \text{(SFVar)}$$
$$\mathsf{clone}(x)[\lceil U \rceil^b / x; s]^a \to \mathsf{clone}(x)[U/x;s]^a \quad \text{(CRed)}$$

All addresses occurring in right-hand sides but not in left-hand sides are *fresh*.

Figure 8: Rules of $\lambda\mathcal{O}bj^{+a}$

The Common Object Module C. This module handles object instantiation and message sending. *Object instantiation* is defined by rule (NO) where an empty object is given an object identity. More sophisticated objects may then be obtained by functional or imperative updates, defined in modules F and I. *Message sending* is formalized by the five remaining rules, namely rule (SP), which propagates the environment into the receiver of the message, rule (SA), which performs the self-application, rules (SU) and (NE), which perform the method-lookup, and at last rule (SRed) which redirects a blocking indirection node. Note that there is no *copy* alternative to rule (SRed), since we still do not want to lose control of the cloning of objects.

The Functional Object Module F. Module F gives the operational semantics of a calculus of non mutable objects. It contains only three rules. Rule (FP) propagates substitutions over functional update operators, installing the evaluation context needed to proceed, while rule (FC) describes the actual update of an object of identity b. The update is not made in place at address b, hence no side effect is performed, but the result is a new object, with a new object identity a which used to be the address of the evaluation context that has led to this new object. This is why we call this operator *functional* or *non mutating*. The last rule (FRed) is the way to get rid of blocking indirection nodes in the case of functional update.

The Imperative Object Module I. Module I contains rules for the mutation of objects (imperative update) and cloning primitive. Imperative update is formalized in a way close to the functional update. Rule (IC) differ from rule (FC) in address management, as illustrated in Section 2. Indeed look at address b in rule (IC). In the left-hand side, b is the identity of an object $[\![O]\!]$, when in the right-hand side it is the identity of the whole object modified by the rule. Since b may be shared from anywhere in the context of evaluation, this modification is observable *non locally* as a *side effect* or *mutation*. Moreover, since the result of this transformation has to be accessible from address a, an indirection node is set from a to b. As described in Section 3, rule (IC) may create cycles because it is possible that the address b is a sub-address of V. Module I has also a rule that redirects blocking indirection nodes in the case of imperative extension, namely rule (IRed).

The term $\mathsf{clone}(x)$ is a primitive for cloning, that performs a lookup in the environment as variable access, but always creates a copy of the found object. As we said before, by copy, we mean a *shallow* copy that creates a new object identity for an existing object even though x and $\mathsf{clone}(x)$ share the same object structure. Rule (CRed) gets rid of a blocking indirection by local redirection.

4.3 Examples in $\lambda\mathcal{O}bj^{+a}$

We first give an example showing a functional object which extends itself [GHL98].

Example 2. Let

$$\mathtt{self\_ext} \triangleq \langle\ \langle\rangle \leftarrow \mathtt{add\_n} = \underbrace{\lambda \mathtt{self}.\langle \mathtt{self} \leftarrow \mathtt{n} = \lambda \mathtt{s}.1\rangle}_{N}\ \rangle$$

The reduction of $M \triangleq (\texttt{self\_ext} \Leftarrow \texttt{add\_n})$ in $\lambda\mathcal{O}bj^{+a}$ is as follows:

$$M[\mathrm{id}]^a \to^* (\langle\langle\rangle[\mathrm{id}]^d \leftarrow \texttt{add\_n} = N[\mathrm{id}]^c\rangle^b \Leftarrow \texttt{add\_n})^a \tag{1}$$

$$\to (\langle\llbracket\langle\rangle^e\rrbracket^d \leftarrow \texttt{add\_n} = N[\mathrm{id}]^c\rangle^b \Leftarrow \texttt{add\_n})^a \tag{2}$$

$$\to (\llbracket\underbrace{\langle\langle\rangle^e \leftarrow \texttt{add\_n} = N[\mathrm{id}]^c\rangle^f}_{O}\rrbracket^b \Leftarrow \texttt{add\_n})^a \tag{3}$$

$$\to Sel^a(O, \texttt{add\_n}, \llbracket O\rrbracket^b) \tag{4}$$

$$\to ((\lambda\texttt{self}.\langle\texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.1\rangle)[\mathrm{id}]^c\ \llbracket O\rrbracket^b)^a \tag{5}$$

$$\to \langle\texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.1\rangle[\llbracket O\rrbracket^b/\texttt{self}; \mathrm{id}]^a \tag{6}$$

$$\to \langle\texttt{self}[\llbracket O\rrbracket^b/\texttt{self}; \mathrm{id}]^h \leftarrow \texttt{n} = (\lambda\texttt{s}.1)[\llbracket O\rrbracket^b/\texttt{self}; \mathrm{id}]^g\rangle^a \tag{7}$$

$$\to \langle\lceil\llbracket O\rrbracket^b\rceil^h \leftarrow \texttt{n} = (\lambda\texttt{s}.1)[\llbracket O\rrbracket^b/\texttt{self}; \mathrm{id}]^g\rangle^a \tag{8}$$

$$\to \langle\llbracket O\rrbracket^b \leftarrow \texttt{n} = (\lambda\texttt{s}.1)[\llbracket O\rrbracket^b/\texttt{self}; \mathrm{id}]^g\rangle^a \tag{9}$$

$$\to \llbracket\langle O \leftarrow \texttt{n} = (\lambda\texttt{s}.1)[\llbracket O\rrbracket^b/\texttt{self}; \mathrm{id}]^g\rangle^h\rrbracket^a \tag{10}$$

In (1), two steps are performed to distribute the environment inside the extension, using rules (SP), and (FP). In (2), the empty object is given an object-structure and an object identity (NO). In (3), this new object is functionally extended (FC), hence it shares the structure of the former object but has a new object-identity. In (4), and (5), two steps (SA) (SU) perform the look up of method `add_n`. In (6) we apply (Bw). In (7), the environment is distributed inside the functional extension (FP). In (8), `self` is replaced by the object it refers (FVar), setting an indirection from h to b. In (9) the indirection is eliminated (FRed). Step (10) is another functional extension (FC). There is no redex in the last term of the reduction.

Some sharing of structures appears in the example above, since *e.g.* $\llbracket O\rrbracket^b$ has several occurrences in some terms of the derivation. However, this example does not show any sharing of computation. The following is a very simple example of a simple "rewriting step" which gives an account of sharing of computation.

Example 3. The addressed term

$$\texttt{x}[\langle\rangle[\mathrm{id}]^a/\texttt{x}; \langle\rangle[\mathrm{id}]^a/\texttt{y}; \mathrm{id}]^b$$

rewrites in one step to

$$\texttt{x}[\llbracket\langle\rangle^c\rrbracket^a/\texttt{x}; \llbracket\langle\rangle^c\rrbracket^a/\texttt{y}; \mathrm{id}]^b$$

by rule (NO). Note how the instance of the new object of identity a is shared by both variables `x` and `y`, which are in fact aliases for this object.

The next example shows a rewriting step performing mutation, then the introduction of a cycle in an acyclic addressed term by an imperative update.

Example 4. The term

$$\texttt{x}[(\llbracket\langle\rangle^c\rrbracket^a \leftarrow\!: \texttt{m} = (\lambda\texttt{s.s})[\mathrm{id}]^d)^e/\texttt{x}; \llbracket\langle\rangle^c\rrbracket^a/\texttt{y}; \mathrm{id}]^b$$

reduces in one step by rule (IC) to $\texttt{x}[\lceil O\rceil^e/\texttt{x}; O/\texttt{y}; \mathrm{id}]^b$, where

$$O \equiv \llbracket\langle\langle\rangle^c \leftarrow \texttt{m} = (\lambda\texttt{s.s})[\mathrm{id}]^d\rangle^f\rrbracket^a$$

Note how the object referred by `y` (in fact `y` becomes an alias of `x`) undergoes the extension

with the new method m, while there was no local operation intended to perform this extension. This is why such a rewriting is called a mutation: from the point of view of y, the referred object has changed—not only syntactically, but also observationally—due to the evaluation of an evaluation context somewhere else in the addressed term.

Example 5. The term

$$\langle \llbracket \langle\rangle^a \rrbracket^b \leftarrow\!: \mathtt{m} = (\lambda \mathtt{self.x})[\llbracket \langle\rangle^a \rrbracket^b/\mathtt{x}; \mathrm{id}]^c \rangle^d$$

reduces by (IC) to

$$\lceil \langle \llbracket \langle\rangle^a \leftarrow \mathtt{m} = (\lambda \mathtt{self.x})[\bullet^b/\mathtt{x}; \mathrm{id}]^c \rangle^e \rrbracket^b \rceil^d$$

This is another example of a loop in the store, easily visualizable by the occurrence of $\bullet$, as the object of identity b contains now a method that references itself. Note how address d redirects to address b where the result of the evaluation context previously assigned address d is stored.

5 Conclusions

We have presented a framework to describe many object-based calculi. To our knowledge, this framework has no equivalent in the literature; it has the following features:

- It is computationally complete since the λ-calculus is explicitly built-in to the language of expressions;
- It gives an account of the delegation-based techniques of inheritance;
- It is compatible with dynamic object extension and self-extension in the style of [GHL98];
- It is generic, due to the partition of rules in independent modules, which can be combined to model (for example) functional *vs.* imperative implementations;
- It supports the analysis of implementations at the level of resource usage, as it models sharing of computations and sharing of storage, and each computation-step in the calculus corresponds to a constant-cost computation in practice;
- It is founded on a novel and mathematically precise theory, *i.e.*, addressed term rewriting systems.

Furthermore, $\lambda\mathcal{O}bj^{+a}$ is generic in the sense that many strategies may be implemented. We have not given any definition of a particular strategy since we do not want to privilege one strategy over another. However, we believe it is interesting to investigate what a strategy is, and how it may be defined. The approach for functional languages studied in [BRL96] should be generalizable to $\lambda\mathcal{O}bj^{+a}$: from a very general point of view, a strategy is a binary relation between addressed terms and addresses. The addresses, in relation with a given term, determines which redexes of the term has to be reduced next (note that in a given term at a given address, at most one rule applies). This is a restriction *w.r.t.* the calculus in which not all the redexes may be reduced. If this relation is a one-to-many relation, the strategy is *non deterministic*. If this relation is a function, then the strategy is *deterministic and sequential*. If this function is computable, then the strategy is *computable*. Implementors and designers

of languages are usually interested in some subclass of the computable strategies, that follows some locality principle—namely that a lot of reductions happen in a small connected part of the whole structure before "jumping" to another distant part. The definition of such strategies—which includes the usual call-by-value, call-by-name, call-by-reference, *etc.*—can be expressed using a very simple set of inference rules (those rule will be collected in another module of $\lambda\mathcal{O}bj^{+a}$ not presented in this paper). These rules can be combined, as basic building blocks, provided possible conditions on their application, to define a lot of strategies.

Finally, we plan to extend $\lambda\mathcal{O}bj^{+a}$ to handle the embedding-based technique of inheritance, following [LLL99], to include a type system, compliant with imperative feature and allowing to type objects extending themselves, following [LLL98, GHL98], and to build a prototype of $\lambda\mathcal{O}bj^{+a}$, from which it should be easy to embed specific calculi and to make experiments on the design of realistic object oriented languages.

A Addressed Term Rewriting Systems

Addressed term rewriting systems (ATRS) [LDLR99] were introduced as a framework which can account for computation with *sharing*, *cycles*, and *mutation*. ATRS's enjoy these features:

- they permit one to model the "geometry" of an implementation: including aspects of sharing and mutation,
- they permit a straightforward representation of cyclic data via "back-pointers";
- they enjoy bounded complexity of rewriting steps by eliminating implicit pointer redirection.

In this sense, ATRS's provide a handy tool for the definition of the formal operational semantics of $\lambda\mathcal{O}bj^{+a}$.

The definitions of this appendix come from [LDLR99] in which the interested reader may find further examples.

A.1 Addressed Terms

Addressed terms are first order terms labeled by operator symbols and decorated with addresses. They satisfy well-formedness constraints ensuring that every addressed term represents a connected piece of a store. Moreover, the label of each node sets the number of successors of the node. More abstractly, addressed terms denote term graphs, as the *largest tree unfolding of the graph without repetition of addresses in any path*. Addresses, noted $a, b, \ldots$, intuitively denote node locations in memory. Identical subtrees occurring at different paths can thus have the same address corresponding to the fact that the two occurrences are *shared*.

The definition is in two stages: the first stage defines the basic inductive term structure, called *preterms*, while the second stage just restricts preterms to well-formed preterms, or addressed terms.

Definition 2 (Preterms).

1. Let Σ be a term signature, and $\bullet$ a special symbol of arity zero (a constant). Let $\mathcal{A}$ be an enumerable set of *addresses* denoted by $a, b, c, \ldots$, and $\mathcal{X}$ an enumerable set of *variables*, denoted by $X, Y, Z, \ldots$ An *addressed preterm* t over Σ is either a variable X, or $\bullet^a$ where a is an address, or an expression of the form $F^a(t_1, \ldots, t_n)$ where $F \in \Sigma$ (the label) has arity $n \geq 0$, a is an address, and each t_i is an addressed preterm (inductively).

2. The location of an addressed preterm t, denoted by $loc(t)$, is defined by

$$loc\big(F^a(t_1, \ldots, t_n)\big) = loc(\bullet^a) = a$$

   It is not defined on variables.

3. The set of variables and addresses occurring within a preterm t is denoted by $var(t)$ and $addr(t)$, respectively, and defined in the obvious way.

Remark 1. Note that in the concrete syntax of $\lambda\mathcal{O}bj^{+a}$ of Figure 7 extends the above scheme in two ways:

1. Symbols in the signature may also be infix (like *e.g.*, $(\_ \Leftarrow \_)$), bracketing (like *e.g.*, $[\![\_]\!]$), mixfix (like $\_[\_]$), or even "invisible" (as is traditional for application, represented by juxtaposition). In these cases, we have chosen to write the address outside brackets and parenthesis.

2. We shall use $\lambda\mathcal{O}bj^{+a}$'s sort-specific variable names.

For example we write $(UV)^a$ instead of $\mathsf{apply}^a(X, Y)$ and $M[s]^a$ instead of $\mathsf{closure}^a(X, Y)$ (substituting U for X, etc.). Indeed, we shall leave the names of $\lambda\mathcal{O}bj^{+a}$ function symbols, such as apply and closure alluded to above, unspecified.

It is clear that not all preterms denote term graphs, since this may lead to inconsistency in the sharing. For instance, the preterm

$$\big((\![\langle\rangle^a]\!]^b \Leftarrow \mathtt{m})^a\, [\![\langle\rangle^a]\!]^b\big)^c$$

is inconsistent, because location a is both labeled by $\langle\rangle$ and $(\_ \Leftarrow \_)$. The preterm

$$\big((\![\langle\rangle^a]\!]^b \Leftarrow \mathtt{m})^c\, [\![\langle\rangle^e]\!]^b\big)^d$$

is inconsistent as well, because the node at location b has its successor at both locations a and e, which is impossible for a term graph. On the contrary, the preterm

$$\big((\![\langle\rangle^a]\!]^b \Leftarrow \mathtt{m})^c\, [\![\langle\rangle^a]\!]^b\big)^d$$

denotes a *legal* term graph with four nodes, respectively, at addresses a, b, c, and d. Observe that computation with this term leads to a *method not found* error since the invoked method $\mathtt{m}$ does not belong to the object $[\![\langle\rangle^a]\!]^b$, and hence will be rejected by a suitable sound run-time type system. Moreover, the nodes at addresses a and b, respectively labeled by $\langle\rangle$ and $[\![\_]\!]$, are shared in the corresponding graph since they have several occurrences in the term.

The well-formedness constraints filter preterms which denote term graphs from preterms which do not. Only the former are called *addressed terms*. The definition of a preterm makes use of a special symbol denoted by $\bullet$, and called a *back-pointer*. The back-pointer is also present in the definition of the syntax of $\lambda\mathcal{O}bj^{+a}$, see Figure 7. The purpose of this symbol is to denote *cycles*. Having a simple representation of cycles is an interesting feature for specifying imperative object calculi, because one can create cycles in the memory by doing imperative updates of objects. Classical rewriting, or algebraic specification tools, lack the provision of a representation of cycles. ATRS representation of cyclic graphs inherits from the work of Rose [Ros96] in using the so-called *back-pointer* representation.

A back-pointer $\bullet^a$ in an addressed term must be such that a is an address occurring on the path from the root of the addressed term to the back-pointer node. It simply indicates at which address one has to branch (or point back) to go on along an infinite path. For instance, the addressed term $[\![\langle\langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda \mathtt{x}.\mathtt{y})[\bullet^b/\mathtt{y};\mathrm{id}]^c\rangle^d]\!]^b$ denotes a cyclic object which refers to itself in the environment and whose cycle originates at address b. Note that $\bullet$ is considered as a special symbol in the sense that it is not a label. In the previous addressed term, the label at address b is $[\![\_]\!]$. Given the previous informal definitions, one could argue that there may be several addressed terms denoting a same cyclic term graph. In fact, there may even be infinitely many. Indeed,

$$
\begin{aligned}
&[\![\langle\langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda \mathtt{x}.\mathtt{y})[\bullet^b/\mathtt{y};\mathrm{id}]^c\rangle^d]\!]^b\\
&[\![\langle\langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda \mathtt{x}.\mathtt{y})[[\![\bullet^d]\!]^b/\mathtt{y};\mathrm{id}]^c\rangle^d]\!]^b\\
&[\![\langle\langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda \mathtt{x}.\mathtt{y})[[\![\langle\langle\,\rangle^a \leftarrow \mathtt{m} = \bullet^c\rangle^d]\!]^b/\mathtt{y};\mathrm{id}]^c\rangle^d]\!]^b\\
&\ldots
\end{aligned}
$$

are just the first three of an infinite sequence of preterms that all denote the same term graph, corresponding to different levels of *unfolding* of addresses b, c, and d. However, it is clear that there is a smallest (with respect to the size of the addressed term) representation of this term graph, namely the first one. In the following, the concept of addressed term will cover only this smallest representations of cyclic term graphs.

An essential operation that we must have on addressed (pre)terms is the *unfolding* that allows seeing, on demand, what is beyond a back-pointer. Unfolding can therefore be seen as a *lazy operator* that traverses one step deeper in a cyclic graph. It is accompanied with its dual, called *folding*, that allows giving a minimal representation of cycles. Note however that folding and unfolding operations have *no operational meaning* in an actual implementation (hence *no operational cost*) but they are essential in order to represent correctly transformations between addressed terms.

Definition 3 (Folding and Unfolding).

Folding. $\mathit{fold}(a)(t)$, where t is a preterm, and a an address, is the *folding of preterms located at* a in t, defined as follows:

$$
\begin{aligned}
\mathit{fold}(a)(X) &= X\\
\mathit{fold}(a)(\bullet^b) &= \bullet^b\\
\mathit{fold}(a)\big(F^a(t_1,\ldots,t_n)\big) &= \bullet^a\\
\mathit{fold}(a)\big(F^b(t_1,\ldots,t_n)\big) &= F^b\big(\mathit{fold}(a)(t_1),\ldots,\mathit{fold}(a)(t_n)\big) \quad \text{if } a \neq b
\end{aligned}
$$

$$
\begin{array}{rcl}
t & \equiv & \langle [\![\langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda \mathtt{x.y})[\bullet^b/\mathtt{y};\mathrm{id}]^c\rangle^e]\!]^b \leftarrow \mathtt{n} = (\lambda \mathtt{x.y})[[\![\langle\,\rangle^a \leftarrow \mathtt{m} = \bullet^c\rangle^e]\!]^b/\mathtt{y};\mathrm{id}]^c\rangle^d \\
u & \equiv & \langle [\![\langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda \mathtt{x.y})[\bullet^b/\mathtt{y};\mathrm{id}]^c\rangle^e]\!]^b \leftarrow \mathtt{n} = (\lambda \mathtt{x.y})[\bullet^b/\mathtt{y};\mathrm{id}]^c\rangle^d
\end{array}
$$

Figure 9: The Addressed Term t and the "not" Addressed Term and u

Unfolding. Let s and t be preterms, such that $loc(s) = a$ (therefore defined), and a does not occur in t except as the address of $\bullet^a$. $unfold(s)(t)$ is the *unfolding of* $\bullet^a$ *by* s *in* t defined as follows:

$$
\begin{array}{rcl}
unfold(s)(X) & = & X \\
unfold(s)(\bullet^b) & = & \begin{cases} s & \text{if } a = b \\ \bullet^b & \text{otherwise} \end{cases} \\
unfold(s)\big(F^b(t_1,\ldots,t_m)\big) & = & F^b\big(t'_1,\ldots,t'_m\big) \quad \text{where} \quad \begin{array}{l} s' = fold(b)(s) \\ t'_1 = unfold(s')(t_1) \\ \cdots \\ t'_m = unfold(s')(t_m) \end{array}
\end{array}
$$

We now proceed with the formal definition of *addressed terms* also called *admissible* preterms, or simply *terms*, for short, when there is no ambiguity. As already mentioned, addressed terms are preterms which denote term graphs. First, we give the reader an intuition of the problems raised by this constraint.

In Figure 9, the preterm t is an addressed term, whereas the preterm u is not an addressed term because the address b does not occur on the path from the root of the term to the second occurrence of $\bullet^b$. Similarly, a sub-term of an addressed term, in the usual sense, is not an addressed term. For instance, $\bullet^b$ is not an addressed term although t, which contains it, is.

This example shows us that the usual sub-term relation is not the one we need. A specific notion of term at a given address *in* an addressed term, abbreviated *in-term*, has the intended property and is given next. This notion tells us that the unique in-term of t located at address b is

$$[\![\langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda \mathtt{x.y})[\bullet^b/\mathtt{y};\mathrm{id}]^c\rangle^e]\!]^b$$

which is an addressed term. Similarly, the unique in-term of t at address c is

$$u = (\lambda \mathtt{x.y})[[\![\langle\,\rangle^a \leftarrow \mathtt{m} = \bullet^c\rangle^e]\!]^b/\mathtt{y};\mathrm{id}]^c$$

although t has three distinct sub-terms at address c, namely u, $\bullet^c$, and

$$(\lambda \mathtt{x.y})[\bullet^b/\mathtt{y};\mathrm{id}]^c$$

The notion of in-term helps to define addressed terms. The definition of addressed terms takes two steps: the first step is the definition of *dangling terms*, that are the sub-terms, in the usual sense, of actual addressed terms. Simultaneously, we define the notion of a dangling term (say s) at a given address, say a, in a dangling term, say t. When the dangling term t (*i.e.* the "out"-term) is known, we just call s an in-term. For a dangling term t, its in-terms are denoted by the function $t@$, read "t at address…", which returns a minimal and consistent representation of terms at each address, using the unfolding.

Therefore, there are two notions to distinguish: on one hand the usual well-founded notion of "sub-term", and on the other hand the (no longer well-founded) notion of "term in another term" or "in-term". In other words, although it is not the case that a term is a proper sub-term of itself, it may be the case that a term is a proper in-term of itself or that a term is an in-term of one of its in-terms; this is due to cycles. The functions $t_i\,@$ are also used during the construction to check that all parts of the same term are consistent, mainly that all in-terms that share a same address are all the same dangling terms.

Dangling terms may have back-pointers which do not point anywhere because there is no node with the same address "above" in the term. The latter are called *dangling back-pointers*. For instance,

$$(\lambda \mathtt{x.y})[\bullet^b/\mathtt{y};\mathrm{id}]^c$$

has a dangling back-pointer, while

$$(\lambda \mathtt{x.y})[[\![\langle\langle\,\rangle^a \leftarrow \mathtt{m} = \bullet^c\rangle^e]\!]^b/\mathtt{y};\mathrm{id}]^c$$

has none. The second step of the definition restricts the addressed terms to the dangling terms which do not have dangling back-pointers.

The following definition provides simultaneously two concepts:

- The dangling terms,
- The functions $t\,@$ from $\mathit{addr}(t)$ to dangling in-terms. $t\,@\,a$ returns the in-term of t at address a.

Definition 4 (Dangling Addressed Terms).

Variables. Every $X \in \mathcal{X}$ is a dangling term. Since $\mathit{addr}(X) = \emptyset$, $X\,@$ is nowhere defined.

Back-pointers. $\bullet^a$ is a dangling term such that $\bullet^a\,@\,a = \bullet^a$.

Expressions. Let $t_1, \ldots, t_n$ be dangling addressed terms ($n \geq 0$) and a be an address such that:

- for all $b \in \mathit{addr}(t_i) \cap \mathit{addr}(t_j)$, we have $t_i\,@\,b = t_j\,@\,b$,
- $a \in \mathit{addr}(t_i)$ only if $t_i\,@\,a = \bullet^a$.

Then, given $F \in \Sigma$ of arity n,

- $F^a(t_1, \ldots, t_n)$ is a dangling term,
- $t\,@$ is defined by:
  - $t\,@\,a = t$,
  - for all $b \in \mathit{addr}(t) \setminus \{a\}$, we have $t\,@\,b = \mathit{unfold}(t)(t_i\,@\,b)$, where t_i is any of $t_1, \ldots, t_n$ containing b.

The "admissible" addressed terms can now be defined as those where all $\bullet^a$ *do* point back to something in t such that a complete (possibly infinite) unfolding of the term exists. The only way we can observe this with the $t\,@$ function is through checking that no $\bullet^a$ can "escape" because this cannot happen when it points back to something.

Definition 5 (Addressed Terms). A dangling addressed term t is *admissible* if

$$\forall a \in addr(t) \text{ we have } t\,@\,a \neq \bullet^a$$

The *addressed terms* denotes the admissible dangling addressed terms.

Proposition 6. *If t is an admissible term, and $a \in addr(t)$, then $t\,@\,a$ is admissible, and $\forall b \in addr(t\,@\,a)$, then $(t\,@\,a)\,@\,b = t\,@\,b$.*

Proof. By definition of addressed terms: it follows from how the function $(t\,@)$ is constructed. □

A.2 Addressed Term Rewriting

Given the representation of term graphs by addressed terms, how do we compute? First of all, the computation on an addressed term must return an addressed term (not just a preterm). In other words, the computation model (here addressed term rewriting) must take into account the sharing information given by the addresses, and must be defined as the *smallest rewriting relation preserving admissibility between addressed terms*. Hence, a computation has to take place simultaneously at several places in the addressed term, namely at the places located at the same address. This simultaneous update of terms corresponds to the update of a location in the memory in a real implementation.

In an ATRS, a rewriting rule is *a pair of open addressed terms*, both located at the same location where an open addressed term is an addressed term which contains variables. The way addressed term rewriting proceeds on an addressed term t is not so different from the way usual term rewriting does. There are four steps.

1. *Find a redex in t*, *i.e.* an in-term *matching* the left-hand side of a rule. Intuitively, an addressed term matching is the same as a classical term matching, except there is a new kind of variables, called addresses, which can only be substituted by addresses.

2. *Create fresh addresses*, *i.e.* addresses not used in the current addressed term t, which will correspond to the locations occurring in the right-hand side, but not in the left-hand side (*i.e.* the new locations).

3. *Substitute the variables and addresses* of the right-hand side of the rule by their new values, as assigned by the matching of the left-hand side or created as fresh addresses. Let us call this new addressed term u.

4. For all a that occur both in t and u, the result of the rewrite, say t', will have $t'\,@\,a = u\,@\,a$, otherwise t' will be like t.

We give the formal definition of matching and replacement, and then we define rewriting precisely.

Definition 7 (Substitution, Matching, Unification).

1. Bijective mappings from addresses to addresses are called *address substitutions*. Mappings from variables to addressed terms are called *variable substitutions*. A pair of an address substitution α and a variable substitution σ is called a *substitution*, and it is denoted by $\langle \alpha; \sigma \rangle$.

2. Let $\langle\alpha;\sigma\rangle$ be a substitution and p a term such that $\mathit{addr}(p) \subseteq \mathit{dom}(\alpha)$ and $\mathit{var}(p) \subseteq \mathit{dom}(\sigma)$. The application of $\langle\alpha;\sigma\rangle$ to p, denoted by $\langle\alpha;\sigma\rangle(p)$, is defined inductively as follows:

$$\begin{aligned}\langle\alpha;\sigma\rangle(\bullet^a) &= \bullet^{\alpha(a)}\\ \langle\alpha;\sigma\rangle(X) &= \sigma(X)\\ \langle\alpha;\sigma\rangle\big(F^a(p_1,\ldots,p_m)\big) &= F^{\alpha(a)}(q_1,\ldots,q_m) \qquad \text{where } q_i = \mathit{fold}(\alpha(a))\big(\langle\alpha;\sigma\rangle(p_i)\big)\end{aligned}$$

3. We say that a term t *matches* a term p if there exists a substitution $\langle\alpha;\sigma\rangle$ such that $\langle\alpha;\sigma\rangle(p) = t$.

4. We say that two terms t and u *unify* if there exists a substitution $\langle\alpha;\sigma\rangle$ and an addressed term v such that $v = \langle\alpha;\sigma\rangle(t) = \langle\alpha;\sigma\rangle(u)$.

Example 6.

1. The term $(\llbracket\langle\rangle^a\rrbracket^d \Leftarrow \mathtt{n})^b$ matches $(\llbracket O\rrbracket^b \Leftarrow m)^a$ with substitution

$$\langle\{a\mapsto b, b\mapsto d\};\{m\mapsto \mathtt{n}, O\mapsto\langle\rangle^a\}\rangle$$

2. The term $\mathtt{z}[\bullet^b/\mathtt{z};\mathrm{id}]^b$ matches $x[U/x;s]^a$ with substitution

$$\langle\{a\mapsto b\};\{x\mapsto \mathtt{z}, U\mapsto \mathtt{z}[\bullet^b/\mathtt{z};\mathrm{id}]^b; s\mapsto \mathrm{id}\}\rangle$$

   Note that the range of the obtained variable substitution consists of addressed terms, as required by the definition of a substitution.

We now define *replacement*. The replacement function operates on terms. Given a term, it changes some of its in-terms at given locations by other terms with the same address. Unlike classical term rewriting (see for instance [DJ90, pp. 252]) the places where replacement is performed are simply given by addresses instead of paths in the term.

Definition 8 (Replacement). Let t, u be addressed terms. The replacement generated by u in t, denoted by $\mathit{repl}(u)(t)$ is defined as follows:

$$\mathit{repl}(u)(X) = X$$

$$\mathit{repl}(u)(\bullet^a) = \begin{cases} u\,@\,a & \text{if } a \in \mathit{addr}(u)\\ \bullet^a & \text{otherwise,}\end{cases}$$

$$\mathit{repl}(u)\big(F^a(t_1,\ldots,t_m)\big) = \begin{cases} u\,@\,a & \text{if } a \in \mathit{addr}(u)\\ F^a\big(\mathit{repl}(u')(t_1),\ldots,\mathit{repl}(u')(t_m)\big) & \text{otherwise, } u' = \mathit{fold}(a)(u)\end{cases}$$

Proposition 9. *If t and u are addressed terms, then $\mathit{repl}(u)(t)$ is an addressed term.*

Proof. By induction on the structure of t as a dangling term. We show more generally that if t and u are dangling terms whose dangling addresses are in any set $\mathcal{A}$, then $\mathit{repl}(u)(t)$ is a dangling term whose dangling addresses are in $\mathcal{A}$. Since an addressed term is a dangling term without dangling address, the intended result follows. □

Example 7.

1. Let t be $\mathtt{x}[\langle\,\rangle[\mathrm{id}]^a/\mathtt{x};\langle\,\rangle[\mathrm{id}]^a/\mathtt{y};\mathrm{id}]^b$, and u be $[\![\langle\,\rangle^c]\!]^a$. The replacement generated by u in t gives $\mathtt{x}[[\![\langle\,\rangle^c]\!]^a/\mathtt{x};[\![\langle\,\rangle^c]\!]^a/\mathtt{y};\mathrm{id}]^b$.

2. Let t be
$$\mathtt{x}[([\![\langle\,\rangle^c]\!]^a \leftarrow\!: \mathtt{m} = (\lambda\mathtt{s}.\mathtt{s})[\mathrm{id}]^d)^e/\mathtt{x};[\![\langle\,\rangle^c]\!]^a/\mathtt{y};\mathrm{id}$$
and u be
$$[\![\langle\langle\,\rangle^c \leftarrow \mathtt{m} = (\lambda\mathtt{s}.\mathtt{s})[\mathrm{id}]^d\rangle^f]\!]^a$$
The replacement generated by u in t gives $\mathtt{x}[\lceil u\rceil^e/\mathtt{x};u/\mathtt{y};\mathrm{id}]^b$.

We now define the notions of redex and rewriting.

Definition 10 (Rule). An addressed rewriting rule over Σ is a pair of addressed terms (l,r) over Σ, written $l \to r$, such that $loc(l) = loc(r)$ (same top address, therefore l and r are not variables), and $var(r) \subseteq var(l)$ (no creation of variables). Moreover, if there are addresses a, b in $addr(l) \cap addr(r)$ such that $l\,@\,a$ and $l\,@\,b$ are unifiable, then $r\,@\,a$ and $r\,@\,b$ must be unifiable with the same unifier.

Definition 11 (Redex). A term t *is a redex* for a rule $l \to r$, if t matches l. A term t *has a redex*, if there exists an address $a \in addr(t)$ such that $t\,@\,a$ is a redex.

Note that, in general, we do not impose restrictions as linearity in addresses (*i.e.* the same address may occur twice), or acyclicity of l and r. However, $\lambda\mathcal{O}bj^{+a}$ is *linear* in addresses (addresses occur only once) and patterns are never cyclic. Cycles may only be introduced by the means of imperative update.

Beside redirecting pointers, ATRS's create *new* nodes. *Fresh renaming* insures that these new node addresses are not already used.

Definition 12 (Fresh Renaming).

1. We denote by $dom(\varphi)$ and $rng(\varphi)$ the usual *domain* and *range* of a function φ.

2. A *renaming* is an injective address substitution.

3. Let t be a term having a redex for the addressed rewriting rule $l \to r$. A renaming α_{fresh} is *fresh* for $l \to r$ with respect to t if $dom(\alpha_{\text{fresh}}) = addr(r) \setminus addr(l)$, *i.e.* the renaming renames each newly introduced address to avoid capture, and $rng(\alpha_{\text{fresh}}) \cap addr(t) = \emptyset$, *i.e.* the chosen addresses are not present in t.

Proposition 13. *Given an admissible term t that has a redex for the addressed rewrite rule $l \to r$. Then*

1. *A fresh renaming α_{fresh} exists for $l \to r$ with respect to t;*

2. *$\langle \alpha \cup \alpha_{\text{fresh}}; \sigma\rangle(r)$ is admissible.*

Proof. (Sketch) The admissibility of t and l ensures that the substitution $\langle\alpha;\sigma\rangle$ satisfies some well-formedness property, in particular the set $rng(\sigma)$ is a set of mutually admissible terms in the sense that the parts they share together are consistent (or in other words, the preterm obtained by giving these terms a common root—with a fresh address—is an addressed term).

The use of α_{fresh} both ensures that all addresses of r are in the domain of the substitution, and that their images by α will not clash with existing addresses.

The definition of substitution takes care in maintaining admissibility for such substitutions, in particular the management of back-pointers. These properties are sufficient to ensure the admissibility of $\langle \alpha \cup \alpha_{\text{fresh}}; \sigma \rangle(r)$. □

At this point, we have given all the definitions needed to specify rewriting.

Definition 14 (Rewriting). Let t be a term which we want to reduce at address a by rule $l \to r$. Proceed as follows:

1. Ensure $t\,@\,a$ is a redex. Let $\langle \alpha; \sigma \rangle(l) = t\,@\,a$.
2. Compute α_{fresh}, a fresh renaming for $l \to r$ with respect to t.
3. Compute $u = \langle \alpha \cup \alpha_{\text{fresh}}; \sigma \rangle(r)$.
4. The result s of rewriting t by rule $l \to r$ at address a is $\mathit{repl}(u)(t)$. We write the reduction $t \to s$, defining "$\to$" as the relation of all such rewritings.

Proof of Theorem 1 from the main text. We wish to prove that the set of addressed terms is closed under rewriting. The proof essentially walks through the steps of Definition 14, showing that each step preserves admissibility.

Let $\mathcal{R}$ be an addressed term rewriting system and t be an addressed term rewritten by the rule $l \to r$. All of t, l, and r, are admissible. For the rewrite to be defined we furthermore know that t has a redex

$$t' \equiv t\,@\,a = \langle \alpha; \sigma \rangle(l)$$

By Proposition 6 t' is admissible and by Proposition 13 we can find a renaming α_{fresh} that is fresh for $l \to r$ with respect to t' and we know that $u \equiv \langle \alpha \cup \alpha_{\text{fresh}}; \sigma \rangle(r)$ is admissible.

Proposition 9 finally ensures that $\mathit{repl}(u)(t)$ is admissible. □

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Massachusetts, 1985.

[Aug84] L. Augustson. A compiler for lazy ML. In *Symposium on Lisp and Functional Programming*, pages 218–227. The ACM Press, 1984.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[BF98] V. Bono and K. Fisher. An Imperative, First-Order Calculus with Object Extension. In *European Conference for Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 462–497. Springer-Verlag, 1998.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[BR95] R. Bloo and K. H. Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *Computer Science in the Netherlands*, pages 62–72, 1995.

[BRL96] Z.-E.-A. Benaissa, K.H. Rose, and P. Lescanne. Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution. In *Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, pages 393–407. Springer-Verlag, 1996.

[BVEG+87] H. P. Barendregt, M. C. J. D. Van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term Graph Rewriting. In *Parallel Architectures and Languages Europe*, number 259 in Lecture Notes in Computer Science, pages 141–158. Springer-Verlag, 1987.

[Car95] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[Cha93] C. Chambers. The Cecil language specification, and rationale. Technical Report 93-03-05, Department of Computer Science and Engineering, University of Washington, USA, 1993.

[Chu41] A. Church. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, N. J., 1941.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers, 1990.

[FF89] M. Felleisen and D. P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69:243–287, 1989.

[FH92] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102, 1992.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[GHL98] P. Di Gianantonio, F. Honsell, and L. Liquori. A Lambda Calculus of Objects with Self-inflicted Extension. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 166–178. The ACM Press, 1998.

[Kah87] G. Kahn. Natural semantics. Technical Report 601, Institut National de Recherche en Informatique et en Automatique, Sophia Antipolis, France, 1987.

[Klo90] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, chapter 6. Oxford University Press, 1990.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6, 1964.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.

[LDLR99] F. Lang, D. Dougherty, P. Lescanne, and K. Rose. Addressed term rewriting systems. Technical Report RR 1999-30, Laboratoire de l'informatique du parallèlisme, ENS de Lyon, France, 1999. Available online at `ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR1999/RR1999-30.ps.Z`.

[Les94] P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In *Principles of Programming Languages*, pages 60–69, 1994.

[Lév80] J.-J. Lévy. Optimal reductions in the lambda-calculus. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.

[LLL98] F. Lang, P. Lescanne, and L. Liquori. A framework for defining object-calculi. Technical Report RR1998-51, Laboratoire de l'informatique du parallèlisme, ENS de Lyon, France, 1998.

[LLL99] F. Lang, P. Lescanne, and L. Liquori. A framework for defining object-calculi (extended abstract). In J.M. Wing, J. Woodcook, and J. Davies, editors, *World Congress on Formal Methods in the Design of Computing Systems*, number 1709 in Lecture Notes in Computer Science, pages 963–982. Springer-Verlag, 1999.

[Mar92] L. Maranget. Optimal Derivations in Weak Lambda Calculi and in Orthogonal Rewriting Systems. In *Principles of Programming Languages*, pages 255–268, 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[Oka98] Ch. Okasaki. *Purely Functional Data Structures*. Cambridge U. Press, 1998.

[PJ87] S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Plo81] Gordon Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

[Plu99] D. Plump. Term graph rewriting. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific, 1999. To appear.

[PvE93] M. J. Plasmeijer and M. C. D. J. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. International Computer Science Series. Addison-Wesley, 1993.

[Ros96] K. H. Rose. *Operational Reduction Models for Functional Programming Languages*. PhD thesis, DIKU, København, Denmark, 1996. DIKU report 96/1.

[Sto77] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Tai92] A. Tailvalsaari. Kevo, a prototype-based object-oriented language based on concatenation and modules operations. Technical Report LACIR 92-02, University of Victoria, Canada, 1992.

[Tof90] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.

[US87] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 227–241. The ACM Press, 1987.

[Wad71] C. P. Wadsworth. *Semantics and pragmatics of the lambda calculus.* PhD thesis, Oxford, 1971.

[WF94] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1), 1994.